1

ADAPTABILITY OF THE LINEAR PROGRAMMING CODES

Wm. Orchard-Hays

P-908 pH

1 August 1956

The RAND Corporation

1700 MAIN ST. • SANTA MONICA • CALIFORNIA

SUMMARY

The operation of an elaborate set of computer codes raises problems of its own which can only be appreciated from experience. Nevertheless, certain general principles for designing such a system can be set forth. The activities which are engaged in during the evolution of such a system are not a simple sequence of events but there is feedback from later steps to earlier ones. How easily the resulting changes can be handled is dependent on the organization of the codes and on the assembly program used. An appendix discusses some shortcomings of the latter together with suggested improvements. ( )

ADAPTABILITY OF THE LINEAR PROGRAMMING CODES

Wm. Orchard-Hays

The preceding discussion* illustrated two points: (i) the operation of an elaborate set of computer codes raises problems of its own which have little to do with the nature of the mathematics, and (ii) if sufficient thought is given to what difficulties can arise and what variations on the method will be required -- both by the formulator and by exigencies of computer operations -- then provision can be made to handle the most common "unusual situations" smoothly.

It is difficult to demonstrate these ideas without getting into the messy details, which we have had to do in several lectures. The operation and use of a large set of codes tends to become an art in itself with considerations which override the very reasons for its existence. One purpose in presenting as many details of our 704 program as we have, has been to furnish an existing -- and we believe a good -- example of a system in actual use with all its attendant paraphernalia.

Nevertheless, there are some general principles that can be stated with regard to the design of such a system. First of all, we note that computers create a lot of work, simply by their nature, which does not exist in hand

* P-909

calculations. Hence the first principle is:

(1) Get the computer to do as much of its own work as possible. This notion is not as obvious as, once stated, it might sound. It has taken four or five years for computer people to come to a firm conclusion in this regard. On punched-card equipment, including the CPC, every effort was made to cut down the work for the machines. But with the high-speed arithmetic and large storage of today's machines, it is a necessity for the computer to do its own assembling of code and data. There is often yet, however, a tendency for a programmer to rely heavily on proven library subroutines and to try to build a code to suit his needs from these. This mode of operation certainly has its place but, for a large class of problems, a compiling routine can be programmed which does the putting together automatically from macro-pseudo-instructions. These latter can often be written by a person with little or no machine experience. One-shot problems or small problems with many parameter combinations or other slight variations are efficiently handled this way. On the other hand, a large program which is designed for a particular type of problem, which is to be used extensively and which pushes against the upper bound of the machine's capacity, must achieve as near maximum efficiency as possible in terms of the machine itself. This means hand-tailored code but, if adequate provision is not made for variations, the resulting program will be too rigid

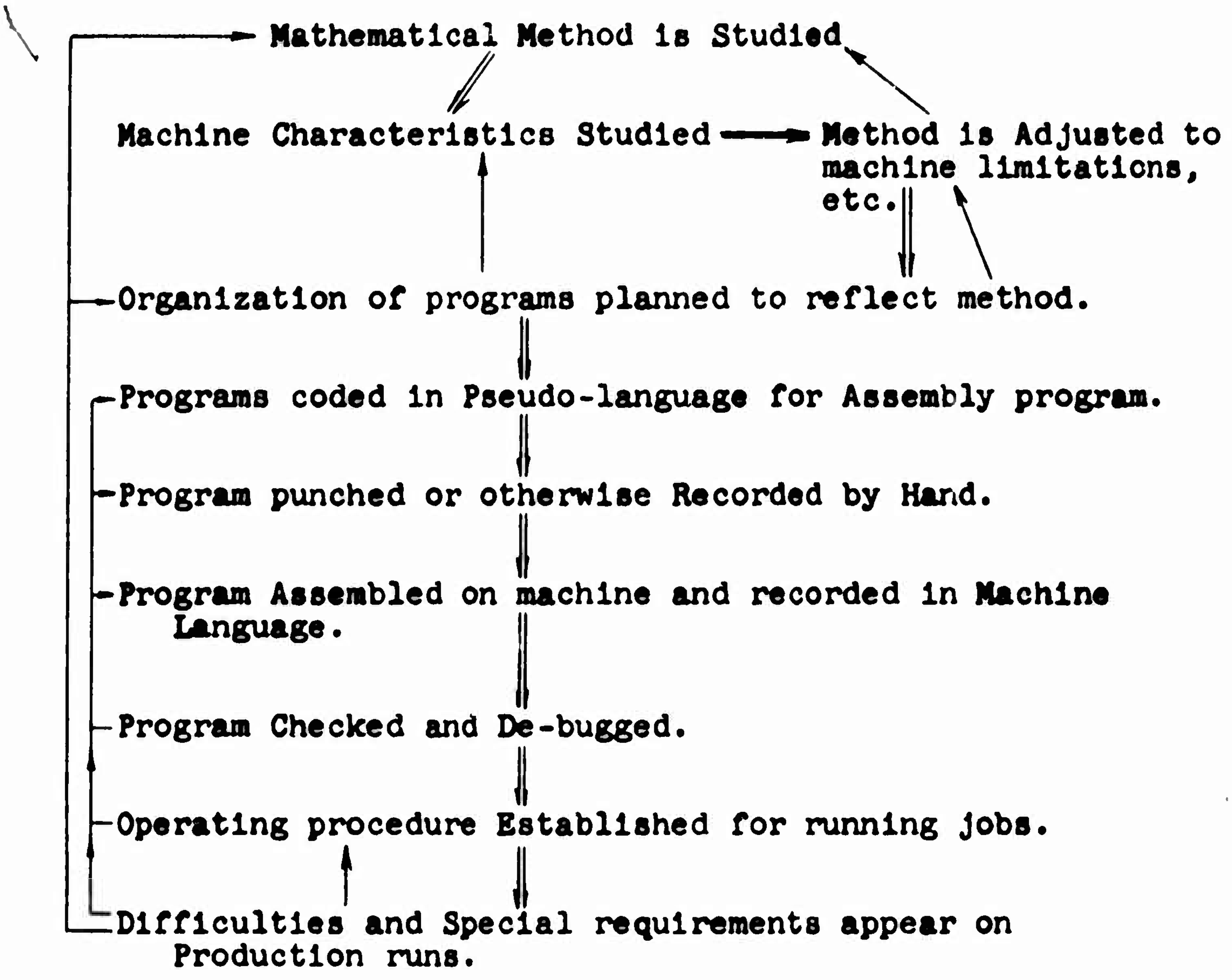
and inflexible. This can be avoided by having the various sections of code reflect the logical breakdown of the method. Hence the second principle:

(2) Adapt the method to the machine and then tailor the code to the method. This will usually rule out the use of standard subroutines. For example, in the L.P. code one might expect to find a standard matrix abstraction program. This is not to the purpose however. An L.P. model handled in standard matrix form is extremely inefficient because of the large number of zeros and, anyway, there are certain required operations on vectors which would not be found in a standard set-up and vice versa. Another common misconception is that the simplex method should be built around a method of solving simultaneous equations. Again this is not to the purpose. The actual solution is, in a sense, the least important part of an L.P. problem. It is the inverse of the basis and the basis headings which are always needed. Our inversion code turns out to be an excellent routine for solving large systems of equations but the converse implication would not be true.

The third principle is more subtle but may be stated as follows:

(3) Make the construction and use of the actual program as independent of time sequence as possible. To better illustrate this point, consider the following events in the creation of a large system of code. The double arrows

indicate the next step and the single arrows indicate feed-back which necessitates modification or re-study.


Mathematical Method is Studied
Machine Characteristics Studied
Method is Adjusted to machine limitations, etc.
Organization of programs planned to reflect method.
Programs coded in Pseudo-language for Assembly program.
Program punched or otherwise Recorded by Hand.
Program Assembled on machine and recorded in Machine Language.
Program Checked and De-bugged.
Operating procedure Established for running jobs.
Difficulties and Special requirements appear on Production runs.

These activities may be iterated on several times before a satisfactory system evolves and the process never really ends. Old codes never die, they just get revised. The question is, in the limit can the process be made to approach zero work? The answer to this is very largely a function of how irreversible each step is and how determinate with respect to its immediate successor. Unfortunately this is controlled to some extent by the assembly program used and, with the increasing tendency toward conformity in the computer profession, we have to live with certain limitations which could be relaxed. But, even so, we can go a long way toward establishing two-way streets everywhere.

Note first of all that we can afford to spend a lot of time on the first three levels. Once the organization is firmed up, we automatically commit ourselves to many conventions and assumptions which must be observed thereafter. That is, once the actual coding is begun, the general organization cannot be changed in particulars without re-examining the whole. Otherwise logical inconsistencies will be inadvertently introduced.

The coding involves a large investment in programmers' time. The hand recording in punched cards or other suitable media is less expensive but facilities for this may be in short supply. At any rate, once a code is punched up, the programmer usually has some hesitance about making major revisions.

The assembly involves a considerable investment in machine time and is singly the most irreversible of the steps in the process. We will have some suggestions later on how this could be improved.

De-bugging requires a large investment both in programmers' time and in machine time. In a practical sense, 100 per cent de-bugging is impossible until production jobs are run because many conditions will never arise in the running of a test case, no matter how elaborate (within reason).

However, the program can be corrected to a point where it is considered checked-out, tentatively. Then an operating procedure must be established. If the programmer was wise, he has looked ahead to this point in planning the organization of the programs but, unless he was a prophet, he has overlooked some things. Awkward handling of equipment may show up and changes will necessitate at least re-assembly, probably some new punching, and possibly considerable new coding. This may all take place, however, within the framework of the basic organization and it is this inner loop which needs to be flexible.

If, in running production, serious difficulties arise and special requirements cannot be accommodated by revising the operating procedure, then one has to go back to the beginning and start over. We have been through this major cycle six or seven times and feel justified in taking some pride in the organization which we have been explaining in the last several lectures. We are, nevertheless, well aware that this organization is inadequate for higher abstractions -- such as block triangularity algorithms -- and also that several improvements are possible within the smaller cycle. There is, however, a considerable amount of inertia to be overcome in making what might seem minor changes to a pure theorist. To see why this is so, let us examine in more detail the coding, assembly, de-bugging and establishing of procedures.

Once one has decided upon an assembler -- or has had it decided for him -- he is committed to numerous rules and conventions which have nothing to do with either the mathematical method or his own organization of the problem for the machine. When the coding is completed -- on paper forms -- it is usually entirely unsuitable for translation to the language of a different assembler and, once punched in cards, is completely fixed. (There is a possibility of partial machine translation between assembler languages in some cases but it is never complete or very satisfactory). But it is still easy to change, add or delete a few cards, provided the usual precautions are observed. However, once we start the next step, we find ourselves irretrievably committed to an expensive process which cannot be interrupted or modified, except trivially, without starting the step over. There is, of course, considerable high-priced machine time involved in assembling a large code. More than this, it is at this point that the complete and up-to-date record of the code is created, where actual references and links are established between parts of the program, and where the massive translation from relative, symbolic, mnemonic, alphadecimal code to absolute, binary machine language is made. Our experience with the L.P. codes has consistently shown that assemblers simply do too much at once for large codes. The tendency today is to build even some features of a compiler into an assembler, in spite of the fact that they serve two distinct purposes. Instead of being tools, assemblers have become black-boxes which gobble up information, spit it out on listings which are non-processible, and then reduce the whole thing to absolute binary cards, which, if wrong, go in the trash can. Until this situation is rectified, it is mandatory to break the program up into independent but compatible parts and to leave some of the putting together to the last possible point in time when an actual problem is being run. The method we have used does this. We carried it farther in our 701 codes by having relative addresses on all instructions which were made absolute only when a section of code was actually to be used during a run, but we found that we got into the region of diminishing returns. In the appendix, we outline an assembler which would overcome most of the difficulties mentioned above but, even with such a tool, it would be profitable to use the same coding philosophy of delaying as long as possible the final commitments on just what parts of the code and what parameters should be used.

The most obnoxious part of the de-bugging process is not so much the finding of the errors and correcting the coding as it is keeping the records, i.e. listings and cards, up-to-date. As one approaches the point of making a system operational, his obligation increases to provide adequate write-ups, listings, forms, procedures, etc. A person who has not attempted this on a large code can have no idea of the trouble it causes. Once program decks, listings and write-ups get out, it is doubly hard to correct them, not to speak of the confusion. This difficulty is considerably reduced by having completely, physically separable parts of the code.

It is when one begins to set up operational procedures and to try to accommodate special job requirements that the "plug-in-unit" concept of organizing a code is most appreciated. If, for a particular run, I would like code No.3, say, to work in a slightly different fashion, I can revise that small piece of code, assemble it, and use the little package of binary cards to replace the corresponding original cards for code No.3. There is no question of it fitting. Of course, it may not work but I know that the errors, if any, are in the little piece I changed. The rest of the program must be right because it is still recorded in the <u>same physical medium</u>. If I had to re-assemble the whole code, then all sorts of things might go wrong which have nothing to do with the intended modification. Of course, the logical interconnections of the various routines must be maintained or the program won't make sense, but there is no trouble with physical interconnections. Furthermore, modification of the logical relationships is rendered easier because these had to be systematically arranged and explicitly stated in the first place in organizing the programs. Patchwork due to afterthoughts are reduced to a minimum and errors found in particular sections of code can be corrected independently of the rest.

In conclusion, let me re-emphasize these points with the notion of remote control of a computer. A modern high-speed, stored program computer is utterly useless as it stands. It is virtually inoperable. A few instructions can be entered manually but, for all practical purposes, every control exercised on the computer, its components, or the data it contains, must be executed by means of a program within the computer itself, including its own loading process. (Hence the term "bootstrap.") Hence it becomes clear that the most important programs are not those which carry out arithmetic functions but, rather, those which assume command of the machine. There is a large payoff in giving these programs great consideration and making them one's alter ego, as it were. The feeling of frustration that one has in walking up to a machine and not being able to get his hands on the things he wants to manipulate can be overcome with these leprechaun routines which do our bidding and control the million-dollar monster which defies us.

APPENDIX

SUGGESTIONS FOR AN ASSEMBLY PROGRAM

We will speak in terms of the IBM 704 but the ideas are equally applicable to any large computer. The first assembly program for the 704 was IBM's NYAP1 assembler (New York Assembly Program 1). It has been almost universally supplanted by United Aircraft's symbolic assembly program (UA-SAP), which is only slightly different but, according to some, an improvement. At any rate, both of these programs accept symbolic instructions, one per IBM card, written in alphadecimal with standard Hollerith punching. Great flexibility is allowed in assigning symbolic addresses or other identification, in making locations relative to arbitrary symbols, etc. In short, the inputs are such as to make the coder's job as easy as possible and to get the machine to do a maximum of organization within the definition of an assembler. (UA-SAP even has some features more apropos of a compiler, as remarked in the main text). We have only one complaint (discussed later) with regard to the rules for programming code which these assemblers accept as input.

Now while quickly admitting great admiration for the coding skill which created these programs, we nevertheless submit the proposition that they are not very handy tools. The trouble is with the output; briefly, they are overly ambitious and seem to have been designed as an end in themselves rather than as a means. When one would like a little ball-peen hammer to smooth out a rough spot, they offer a rolling mill. Or, instead of a proof sheet, they hand one a bound book (well, almost).

Yet, one cannot operate without an assembler. The symbolic code for the L.P. programs takes about a tray and a half of a standard filing cabinet or about 4800 cards. Imagine trying to code and record these in absolute binary! The assembler processes these and produces about 225 binary cards -- 22 absolute, machine language words per card. (Some symbolic instructions produce more than one absolute word and not all binary cards are full.) At the same time, the 4800-odd instructions are printed out in both symbolic and absolute octal. This is the primary document for the program. This printing is a very expensive operation and, if wrong, must be done again, at least in part. Remember that we have not yet begun de-bugging. If the printing is right, it is still expensive to reproduce.

If now a mistake is found in a section of code, it must be corrected and all the original symbolic cards for this section assembled again. The binary cards produced the first time are junked and the new listing must be fitted in to replace the original incorrect part. (One simple improvement in present assemblers would be to start a new page on each "origin" card, i.e. at the start of each new section.) Not only is there no salvage from the first assembly, but the correction of the primary records is awkward.

Now the time-consuming part of an assembly is not the assignment of actual locations and punching of binary cards. This is fairly cheap. It is the reading and printing of original symbolic code that is expensive. But what if a symbolic card is wrong? While this may invalidate the entire output, it usually does not affect the other symbolic cards. Probably 97 to 98 per cent of these are right the first time. When the assembler reads these cards it converts the information to a binary-coded alphadecimal form which can be packed into much less space. But, at this point, no information has been lost, it has only been written more efficiently. If the code being assembled were saved in this status, in a form which could be hand corrected, re-assembly would be cheap, except for printing. But again, what of printing? Do we really have any use for all those reams of paper in de-bugging? The answer is no, in fact we would be better off with much less information given in condensed form. The detailed information is available on the binary cards if it is really required. When the code is checked out, then is the time for a full and complete -- and correct-- listing of the program.

Aside from ambiguities and mispunched hash detected by the assembler, the really useful information in de-bugging is the absolute location assigned each symbol. However, one feature of present assemblers makes every word of code a potential symbolic location. This is the feature of addresses relative to a symbol, e.g. SYMBL+6 means the sixth address after the address assigned to SYMBL. Similarly ABCDE-2 means the second address before the one assigned to ABCDE. This seems like a marvelous convenience from a coder's viewpoint. We found, however, in de-bugging the LP codes, that the great majority of errors occurred from miscounting the addresses from the symbol or forgetting to correct after insertions or deletions near the symbol. The references are often far removed. Of course, the obvious way to avoid this is to use a distinct symbol for each referenced location. However, this increases the amount of information required by the assembler and it is often convenient for symbols in a sub-section of code to be related. For example, one might use SYMB, SYMB1, SYMB2, etc. for referenced locations near together. This in fact is done throughout the coding of UA-SAP itself! (Amazingly, it is written in terms of itself.) Hence, why have relative locations at all? A better device would be second-order symbols, i.e. sub-symbols which are reheaded automatically whenever a new major symbol appears, e.g.

(Location)		(Reference)
SYMB -	=	SYMB
1	=	SYMB1
6	=	SYMB6
ABCD -	=	ABCD
3	=	ABCD3

where the location sub-symbols are written in a distinct column set aside for them. With these remarks, we can now state a set of proposed specifications for an assembly program. We will assume familiarity with UA-SAP.

We would accept the standard "SHARE" 3-letter symbolic operation code in its entirety including the additional "prefix operations" MZE, MON, etc. We would modify the UA-SAP definition of a symbol to read as follows:

SYMBOL:	Any combination of not more than 5 Hollerith characters except the comma, the first of which is non-numeric.
SUB-SYMBOL:	A symbol prefixed with a digit 1, ..., 9.
FIXED SYMBOL:	A symbol prefixed with a minus sign.

We would reserve column 1 for a symbol prefix but otherwise adopt the UA-SAP symbolic card form.

We would do away entirely with arithmetic expressions in terms of symbols and integers, including relative addresses. (These are properly functions of a compiler.)

We would require all tags to be numeric and disallow all-numeric location fields to act as origin cards.

We would accept the UA-SAP pseudo-instructions ORG, EQU, SYN, DEC, OCT, BCD, BSS, BES and END as defined. The EQU instruction would require a decimal address and the SYN operation a previously defined symbolic address.

We would omit entirely REP, LIB, HED, DEF and REM and impose some restrictions on remarks. We would also add the pseudo-instructions GRP, OPE and CLR which will be explained subsequently.

From this point, we diverge from UA-SAP, for we would introduce "circulating information" in the form of BCD (binary coded decimal) punched cards. All symbolic cards (with certain exceptions) would be punched in this form before punching the absolute binary deck. These BCD cards would be readable by the assembler and could be intermixed with symbolic cards for re-assembly. The BCD cards used as input would not be re-punched.

These BCD cards would be punched in groups, each group being associated with a major symbol. The first card of a group would be distinctive and be associated only with the major symbol. We shall hereafter refer to these as S-cards. These are the only BCD cards on which comments would be reproduced and only the comments on symbolic cards with a major symbol in the location field would eventually be printed. The original alphadecimal symbol would be punched in standard Hollerith characters in columns 73-77 (no prefix) on an S-card which would act as a gang master for the rest of the group. (If an automatic sequencing device is available on the card punch, this could be used on columns 78-80). The BCD cards could then be interpreted for easy identification.

The other cards of the group will be referred to as G-cards. These would contain up to 12 instructions each. Instructions which contain neither a location nor a decrement would require one row of a card, otherwise two rows.

Symbolic cards with DEC, OCT or BCD operations and a major symbol in the location, would be reproduced on one S-card with no comments.

If it should be desired to use several major symbols in sequence, as for a region of universal constants, then it would be desirable to punch them all in G-cards. This one could do by heading the symbolic cards with a card containing the pseudo-instruction GRP. All following symbolic cards would then be reproduced in G-cards (with no comments) until either a symbolic card with the instruction OPE or an S-card was read. Columns 73-77 of these cards would retain whatever symbol was previously being ganged.

Symbolic cards with the pseudo-instructions ORG, EQU, SYN, GRP, OPE, CLR, or END would not be reproduced in BCD cards since this information might well vary from assembly to re-assembly.

The pseudo-instruction CLR would accomplish the following:

All unfixed symbols are sorted, stored on tape and cleared out of the symbol table.

This would eliminate the difficulty of duplicate symbols in different sections of the program being assembled and be much more to the point than the UA-SAP instruction HED. Universal symbols (which ought to be universally honored) could be fixed. When the END card is read, all symbols remaining in the table would be sorted. Since the fixed symbols would be negative, they would occur first. The remaining unfixed symbols would be written on tape which would then be rewound and the records of unfixed symbols, i.e. sets divided by CLR cards, would be used in order of occurrence together with the fixed symbols.

The BCD cards would be punched during reading of the symbolic cards. On the first assembly, this would require considerable punching but, on reassembly, only corrected symbolic cards would be punched and reading time should be cut by a factor of six or eight.

There would be two kinds of printing, each subject to a switch. Before each set of symbols is sorted, they could be printed out in order of occurrence together with the absolute locations assigned. This would be the only printing normally done on a first assembly. The other option would be the complete listing of the assembled program in a form entirely similar to the UA-SAP format. This could be deferred until the program was de-bugged. At that time only BCD cards would have to be read (none punched) and printing could be done on vellum for reproduction.

The assembled program would, of course, be punched in absolute binary cards in any case.

The detailed formats of the S- and G-cards have been worked out and are entirely practicable. They are not discussed here as being of interest only in the event such a program is developed. This would, of course, be a two-pass assembler. The program for the first pass has been outlined and a considerable part of it actually coded. The second pass (actual assembly) would not be much different from the second pass of UA-SAP. On the first pass, there is no question of being able to handle 12 instructions per G-card at 250 cards per minute since there is no conversion, packing, editing, etc. to be done. This would have all been accomplished on the first assembly.